

Virtual Memory Within Embedded Systems - Marketing Hype or Engineering Reality?

This paper begins by addressing the operation of virtual memory in general. It proceeds with the benefits of the use of virtual memory in a real-time system such as one using the OnCore microkernel and contrasts this with other, less complete, approaches to memory resource utilization. The various approaches to memory utilization in an RTOS are categorized and compared. A results summary is presented.

HOW VIRTUAL MEMORY WORKS

Virtual memory is a methodology for permitting different processes (application programs) to share the memory and processor resources of a computing system or platform. This sharing is protected so that a failure, such as an attempt to write to an illegal address in one of the tasks will not result in the failure of any other tasks. This is exactly the same state-of-the-art protection and partitioning behavior as is available and expected in a Unix/Linux system. It permits new tasks to be started and to execute along with the pre-existing tasks without the need to make any modifications to those existing tasks or for tasks to even be aware of each other.

The implementation of virtual memory depends on a hardware Memory Management Unit (MMU) and on support from the operating system to properly utilize this MMU. The MMU provides the ability to map (or translate) memory addresses (from logical to physical addresses) and to protect memory locations from illegal accesses. The mechanism that permits a program to run in any location in physical memory is called relocation. The hardware implementation of these capabilities in the MMU is the only way to provide these facilities in a secure and efficient fashion.

Most modern processors include an MMU as an integral part of the processor. These processors have their internal designs optimized to operate in a virtual memory environment. These modern internally implemented MMUs no longer suffer from the memory access delays that were obvious and necessary when an external MMU was added to an older processor such as a 68000. The MMU is now designed to be an integral part of the processor's instruction execution pipeline(s). Not only is there no longer a penalty for utilizing an MMU, there will likely be processor inefficiencies introduced if the system software designer elects not to use the MMU. In some cases disabling the MMU, thus by passing the processor's intended pipeline, can cause the processor efficiency to drop by over 80 percent.

Memory mapping refers to the process of translating the set of addresses used in a task (logical addresses)

to those representing the actual physical memory locations (physical addresses) that are used. Each task utilizes memory to store its

1. instruction code (binary machine instructions, called "text" in Unix/Linux),
2. statically allocated data (e.g. static or global variables),
3. stack or dynamically allocated data used in a last in, first out (LIFO) manner for example with dynamic variables and return addresses, and
4. heap, for data storage locations allocated and released as specified by the programmer.

Each of these four regions of a task is partitioned into equal-sized pages. A typical page size for a modern processor is 4 Kbytes (4096 bytes). Although current processors may actually use more than one page size, page sizes are always a multiple of a power of 2 to greatly simplify and streamline the actions that must take place in the MMU to perform the mapping. When a page is placed in physical memory it is placed into a page frame. A page frame is a section of physical memory that is the same size as a page and located at an address that is an integer multiple of the page size. An integral (whole) number of pages is always allotted to each of the above regions. The MMU contains or accesses tables that must be loaded with the appropriate logical address to physical address mapping. The operating system is able to change this mapping as required as a task's memory requirements change or when tasks are added to or deleted from the system. The operating system also has the responsibility to change this mapping each time a task switch occurs. Let's take a look at how the address translation works. As an example, we'll assume a system with 32-bit addresses and 4096-byte pages. For a system such as this, we can then consider each 32-bit address to be a combination of a 20-bit page number (at the most significant end of the address) and a 12-bit offset (at the least significant end of the address) into that page. The MMU will examine the 20-bit page number and provide (via mapping or translation based on the MMU tables) a page frame address for this page. By concatenating this translated address (the page frame

MEMORY MANAGEMENT

address) with the original 12-bit offset, the resulting physical address is produced. A typical modern pipelined processor may allocate a pipeline stage for this MMU translation and another stage to then actually access the memory.

Each page of memory also has various privileges and status information associated it. The MMU dedicates a bit to indicate each privilege or status item for each page. These bits are used to indicate whether the contents of the page:

1. can be used as processor instructions (execute privilege),
2. can be written (write privilege)
3. can be read (read privilege)
4. have been written to (dirty bit)
5. are currently present in physical memory (valid bit)

By utilizing virtual memory, it is possible to execute tasks whose memory requirements actually exceed the total amount of physical memory available in the system. When demand paging is added to a virtual memory system, a memory page accessed (legally) by a task will be brought into the physical memory automatically at this first reference. The reference to a page not currently in physical memory causes a special exception called a "page fault" to be generated. The operating system is then responsible for retrieving the requested page from a "backing store" such as a hard disk or FLASH memory and restarting the task at the same machine instruction that generated the page fault.

REAL-TIME PROCESSING AND THE ONCORE MICROKERNEL

Obviously, demand paging cannot be used for tasks that have hard real-time requirements because of the obvious unscheduled delay involved with page faults. However, by using a modern microkernel technology, such as that offered by OnCore Systems, it is possible to have tasks with hard real-time scheduling requirements as well as other tasks that use demand paging to be running on the same system. This is simply accomplished by using OnCore's `vm_wire` system call to wire down a real-time application into physical RAM. These pages are then "wired down", a physical metaphor to indicate that they are not to be removed from physical memory. Within the OnCore microkernel, the `vm_wire` function can be used to wire or to unwire a region of virtual memory. The wiring and unwiring will always correspond to the set of whole pages that correspond to (contain) the memory region to be affected. Access to this capability is only available via the privileged host port of the host on which the target task executes because of the privileged nature of an operation that affects the physical memory resources in this way. Prior to using the `vm_wire` function call, information about the use of memory by the task can be obtained by using either the `vm_region` or the `vm_statistics` function call.

Whenever a task attempts to reference a page for which it does not have the appropriate access privileges or for which it has no access rights at all, an

exception is generated and the microkernel is notified. The protection and privileges that this implements include for example:

1. it is impossible for any task to write into or read from those portions of physical memory not allocated to that task.
2. it is impossible for a task to write to the portion of its memory space designated as executable.
3. it is impossible for a task to write to portions of memory that are to be read-only such as static strings, initialization data etc.
4. bad pointer use can be detected and intercepted as it occurs rather than later when use of the corrupted memory or bogus data might be attempted

The benefits of using virtual memory in enterprise computing systems are very well known and documented. As embedded real-time systems become more sophisticated, many of these same benefits become attractive in this arena as well. Modern embedded systems have a mix of tasks, each with its own resource requirements and timing constraints. Furthermore, high-availability (HA) systems must be able to seamlessly mutate this mixture of tasks to meet such changing needs as software upgrades and performance monitoring.

A virtual memory system permits each task to execute in its own virtual address space (i.e. it can be compiled and linked to execute at any address). It need not even be aware of the other tasks that are executing on the system. The system will assign the pages that comprise the application to locations in physical memory that are not presently in use by any other task. There is no need for these regions to be contiguous, only that physical memory be large enough to contain all the desired applications (assuming a fully real-time system that uses no demand paging). In a system with demand paging, adequate physical memory for efficient execution (i.e. to contain the working set) of these non-real-time tasks must also be provided.

In a system designed to require separate tasks to communicate for the purpose of sharing data, virtual memory enables the sharing to be done efficiently in a totally protected manner. Virtual memory techniques are especially efficient when a large amount of data (e.g. lengthy messages) must be passed from one task to another. In the case of OnCore Systems, it's the microkernel which permits a zero-copy transfer of data and Inter-Process Communication (IPC) between fully protected domains. The data to be passed is prepared in a page that initially is owned with write access privileges by the sender of the data. The act of sending removes access privileges from the sender and provides read access to the receiver of the message. The message area is fully protected during this entire process and the transfer is high performance because there was no need for the operating system kernel to copy data as would be required in a system that does not implement a true virtual memory system. Also note that some other vendor's systems require a copy operation during message passing but perform the actual copy during context switch operations. This produces a system that appears to have very rapid message

passing but at the expense of large and unpredictable context switch times.

The virtual memory system in the OnCore kernel consists of machine dependent and machine independent portions. The machine dependent portion is responsible for manipulating the special processor hardware features used to set access permissions and to communicate the intentions of the machine independent portion to the processor. The machine independent portion is responsible for mapping out the virtual address space, setting memory ranges within this map and for an interface to the (optional) backing store. Demand paging with its backing store are not a part of the OnCore kernel, but are added by a separate task such as the Linux kernel when Linux is run above the OnCore kernel. User tasks can provide backing storage for memory ranges as an application may demand. Full support for advanced computer architectures like Non-Uniform Memory Access (NUMA) or for virtually mapped caches has not yet been incorporated into the kernel.

A virtual address space (the set of addresses that are legal for a task to reference) is always owned by the task. Any thread executing within the task has access to all the addresses (as well as other resources) that are owned by that task. Inside the kernel, memory is seen as divided into regions (groups of virtually contiguous pages), which are then controlled by the kernel. Each region has a consistent set of attributes (protection, inheritance characteristics etc.) When virtual memory is completely implemented as it is in the OnCore OS, the following features are all available:

1. A program can be loaded into any (not necessarily contiguous) portion of the physical memory. For non-real-time processes, the memory image can later be swapped to disk (copied to disk and the RAM reused for other purposes). Later, when the memory image is returned to RAM to continue execution, it may be placed in a completely different region of the physical memory. After this happens, the logical address space (the program's own view of memory) has not changed, however, the physical address space has.
2. Multiple programs, each in various states of execution, can reside in physical memory at any time. The hardware-enforced permissions are used to ensure that no program can have any access to regions of memory belonging to any other program. This is an especially valuable feature because it can prevent a software error (a bug) in one program (e.g. a bad pointer) from corrupting another program.
3. Within any program there are portions of memory that should only be accessed in pre-determinable ways. The portion of memory containing the machine language instructions for the processor (called text in a Unix system) should be used for that purpose only and should not be read from or written to as a data area would. Initialization data and constants should only be read, and never written to. Other portions can be read or written when the processor is manipulating data. The virtual memory system can enforce these regions as well and can prevent some very elusive bugs from modifying regions of memory improperly.
4. Programs can share data with one another in an unprotected or in a completely protected fashion. Virtual memory supports efficient protected sharing of information. With properly designed message passing support in the operating system, a message can be composed within one or more pages within the address space of the sender. The action of sending the message causes the operating system to make this page or pages available to the recipient of the message. Notice that it was unnecessary for the operating system to copy the message from one protected space to another protected space to maintain this security. This is known as zero-copy messaging. A follow-on to this capability is the implementation of copy-on-write (COW). With copy-on-write, multiple processes (programs) can retain read access to a page or set of pages. When any such process, however, writes to this area, a private copy of the area is created for that process by the operating system and is mapped into the same logical address space where the shared data had appeared in the logical address space. This method is much more secure and produces systems with higher reliability than using a traditional shared memory approach.
5. Programs can also share executable portions (rather than data portions). Shared libraries are designed to utilize this capability. The physical memory containing the shared libraries can be mapped into the logical address space of each process that needs access to these code libraries. Since this region can be protected in such a manner that none of the programs using these shared libraries can modify them, this is a safe and efficient utilization of physical memory.
6. In the virtual memory systems of engineering workstations, demand paging is always available and used. Demand paging refers to the facility for not requiring an entire program to be present in physical memory as it is executing. Whenever a legitimate access is made to an area of program or data (access to code or data that has been assigned as belonging to that program), the MMU checks to see if that logical address is presently mapped to a physical address (i.e. present in RAM). If it is, the mapping happens, the physical memory is accessed and all proceeds normally. If, however, the logical address is not presently mapped to a physical address, an exception called a "page fault" occurs. The page fault is immediately handled by the operating system (OS). The action taken by the OS is to copy the page containing the requested logical address into an available page frame, set up the proper mapping information in the MMU and to restart the machine language instruction that caused the page fault. If no page frame is available when the page fault occurs, the operating system is tasked with finding some other page to remove from memory to make room for the page that has been requested now. This transfer of data between memory and the backing store is called "swapping".

MEMORY MANAGEMENT

The operating system will usually use a replacement algorithm such as the Least Recently Used (LRU) algorithm to determine which page should be removed from memory. The MMU also maintains a dirty bit for each page in physical memory. The dirty bit is used to indicate if the page in question has been written to (changed, and therefore no longer identical with the page copy on disk). If it has, it will be necessary for the operating system to first copy that page to the disk before bringing the requested page into physical memory.

7. Some pages can be designated as wired down (using OnCore's `vm_wire` system call). Such a page is not a candidate for removal from physical memory by swapping. Pages necessary for processes that have hard real-time response requirements must be wired down. This avoids the non-determinism that would result if a page fault might be required to access the needed information.

OTHER APPROACHES

Systems that attempt to build a somewhat virtual memory-like environment on top of a more traditional RTOS have also recently been created. These hybrid systems create memory partitions that can be used to protect one task from another, but invariably need to provide some form of protection/efficiency trade-off. Because the protected transfer of a message from one task to another is more time consuming than an unprotected transfer in these systems, the software developer is provided with the option to disable the protection, and hence improve the speed of data transmission. A commonly promoted practice with these systems is for the system developer to test his/her code in the protected mode and then run it in the unprotected mode to improve its performance in production environments. A task constructed in this manner will certainly result in an overall system that is much more difficult to validate, or worse, might result in a system that fails inexplicably. A system with a true virtual memory implementation can compartmentalize the code by erecting access barriers around each task. This permits constructing a system that can be more easily validated and certified by permitting each task to be individually validated. The resulting testing regimen is much simpler than when these access barriers are not present in the deployed system.

Hybrid systems like those noted above do not have a well-established history of provably correct and complete protection. They may be easier to use and configure than those using a traditional RTOS, but they also represent potential unnecessary complications in performance and scalability relative to a system based on a complete implementation of virtual memory and employing zero-copy messaging. The user model of interacting with these hybrid systems is typically less complete, more complicated and subject to improper use by application developers who may not fully understand the protection model and its proper use. Systems that rely on the application developer to decide how each application should be partitioned and protected are less likely to be as secure as systems where the protection is inherent in the underlying

system. The scalability of the hybrid systems has not yet been proven, whereas the scalability of a true virtual memory system has been proven and implemented for decades.

Again, these hybrid systems appear to provide only a subset of the capabilities that a true virtual memory approach like that provided by the OnCore microkernel provides. Because the OnCore microkernel provides an OS-neutral abstraction consisting of a virtual memory-based platform on which other operating systems can be hosted, it is possible to run Linux, as well as other operating systems simultaneously and to provide secure communication and task switching with real-time response across each of these operating systems. Because many RTOSs are now running on modern, cached, super-scalar microprocessors with integrated MMUs, most are now trying to make the difficult transition to a more robust environment supporting simple MMU-based application partitioning. Because most now support a bolt-on MMU strategy that forces them to perform simple address translations into static memory partitions, they are claiming to use virtual addresses, and therefore have virtual memory. This makes for good marketing collateral, but simply is inflating the use of traditional computer science terminology.

RTOS MEMORY MODELS

In the RTOS/embedded space, there are four (4) basic levels of RAM-based "virtual memory" usage.

- **VM Level 0** - This is the "flat" memory model of the Motorola 68000 family - there is no virtual memory whatsoever. All available memory is accessible by any program or the operating system. There is no Memory Management Unit (MMU) usage in VM Level 0. Data is typically passed between threads by passing a pointer.
- **VM Level 1** - This is the model that many RTOSs, conceived in the 68000 era, use to cope with newer microprocessors with integrated MMUs and advanced memory caches. The designers of these super-scalar, pipelined-architecture chips assumed (and rightly so) that the software on these chips would use true virtual memory, so they require the MMU and cache to be enabled. Because most legacy RTOSs are dependent on a flat memory model for correct operation, most simply turn on the MMU and create one partition. Merely turning on the MMU and thus performing the (required) address translation on every memory access is definitely not "virtual memory", although some vendors are using this term "virtual memory" incorrectly. It is instead, simply the flat memory model using 1-to-1 memory mapping with the MMU turned on. All 68000-class RTOSs running on newer silicon support this model.
- **VM Level 2** - This is the model where the MMU is turned on and static memory partitions are created to protect each (or some) application and the OS, by causing illegal access exceptions when pointers try to access another program's address space. Once again, many vendors call this "virtual memory" - it is

not. Many vendors also call this "safe" - it may not be so, depending on the implementation. Within this memory protection model, moving data between MMU-protected partitions is typically done using a message-passing system that essentially copies the data into the target partition. Sometimes this copy is performed during context switch time. Green Hill's Integrity, Enea's OSE, and Wind River's new AE platform are all in this category.

- **VM Level 3** - This is true virtual memory. Pages of memory are dynamically allocated, de-allocated, and re-allocated on the fly by the operating system using a sophisticated memory mapper that works efficiently with the address translation facilities of the CPU. The transition from a pure RAM-based memory to an optional disk-based demand paged virtual memory (backing store) is transparent. Messages and data are mapped automatically to programs as required, and some vendors (like OnCore) have a sophisticated copy-on-write technology to efficiently control simultaneous writes to the same data space. Programs are loaded and mapped into a dynamic address space as required. This environment is a natural fit for creating virtual machines and MMU-protected separate address spaces for running VM-encapsulated applications and/or operating systems. This is true virtual memory. This is the OnCore solution. This is also the type of virtual memory used by Sun's Solaris, Apple's OSX and other world-class OSs supporting mission-critical environments.

SUMMARY

High availability without true virtual memory is not a viable solution. A fundamental requirement of a true high availability system is a Level 3 virtual memory system. The fundamental problem in systems with less than VM level 3 support is that as programs get updated and/or failed over, gaps in the static memory partitions occur, and a full update of the system is required at some time. With a true virtual memory solution (Level 3), every page of memory is dynamically mapped on demand, so there is no wasted space that cannot be automatically recovered by normal operations. A fundamental cleanup will never be required. As a summary, lets compare a system with VM Level 3 (true virtual memory) to a system without this capability.

With true virtual memory, zero-copy message passing is available. De-allocating memory buffers from the sender's protected address space and allocating the buffers to the receiver's protected space provides true zero-copy message passing. This is a unique feature of dynamically allocated memory systems. Protection does not need to be turned off to improve the performance of message passing.

With true virtual memory, managing the virtual address space is an integral part of the operating system. Managing page tables and protected partitions is deterministic, it is simply a save and restore operation in a true VM system. Static memory systems, with add-on or bolt-on protection will need to walk page tables to change permissions, a non-deterministic process.

With true virtual memory, code is automatically position independent. Every task is built to run at logical address zero (or any other address) no matter where the code is physically located. This means:

1. Binaries are portable from system to system. This is unique to dynamically allocated memory systems.
2. Testing time for applications is reduced because the physical code placement is not a factor in testing. Testing of every permutation of a system is not required since code runs from logical addresses.
3. Page zero can be protected by building code to start at an offset from page zero, therefore, no physical page is wasted.
4. Various configurations and options are simpler to manage and release, and can easily be upgraded in deployed systems.

With true virtual memory, updating live systems is simpler. New or updated applications can be loaded and started without affecting existing systems, even in a live system. Since the operating system manages the dynamic allocation of memory, programs may be loaded anywhere in memory. Contiguous blocks of memory are not required. Since applications run in a logical address space, physical location is not a factor. Removing (killing) a program is just as easy. The (physical) memory is automatically recovered by the system. Repairing a live system is simplified because:

1. Tasks or applications that fail may be reloaded and restarted without affecting other applications
2. Warm restarts of failed or impaired applications is simplified by reloading the application then copying any valid state information to the new task.

True virtual memory efficiently uses memory space. In a RAM based system, free memory is a list of pages. The position of pages is not a factor. No recovery of fragmented memory is ever required, not during execution of normal applications or after updating the system. A static memory system must explicitly recover fragmented memory, especially after multiple software updates and patches. This will likely require a complete reload of the software.

True virtual memory allows each task to run in its own virtual machine. Each task appears to be the only thing running in the environment. This enables operating systems like Linux to be run in their own virtual space. It also allows JAVA to be run without consuming the entire machine. This is a major factor that enables the portability of applications and is not available on static memory systems.

True virtual memory enables multiple environments to run independently as if on separate processors. This includes running Linux applications and real-time applications simultaneously or even running multiple Linux operating systems simultaneously.

True virtual memory allows a variety of safety critical or quality critical systems to run on the same physical system.

1. Low quality or low use software can reside safely with critical software since each is fully protected in its own virtual space. Illegal memory accesses can-

MEMORY MANAGEMENT

not effect other tasks in the system.

2. The testing of non-critical software does not need to meet critical software testing standards, since they are fully protected from each other. This can save significant time and money on the testing of non-critical software.
3. Third party software may be added to a system without fear of unknown failure due to rogue software illegally writing to and corrupting critical system areas.

True virtual memory permits scaling to multiple processors in live systems. Adding an additional processor, and moving an application from one processor to another is possible because of the binary portability of applications. Along with transparent message passing features of the Inter-Process Communications (IPC) system, messages will be redirected to the application on the new processor. The sending application does not need to change, or even be aware that the target application has moved. The sending application is passing the data to a Named Port, and the operating system sends the message to the correct location.

True virtual memory allows the transition from a pure RAM based system to a disk based, demand paged system to be transparent. A mixture of real-time and non real-time applications can be placed in the same system. Non real-time applications may even be demand paged.

Note: In the continuing saga of inflating traditional computer science terms, a few RTOS companies that now have a MMU option are also claiming that they are Fault Tolerant. Fault Tolerant in this inflated sense means that they are "somewhat tolerant of some fault conditions". The true definition of fault tolerance is the traditional "Tandem-class" computer solution - a close coupling of redundant hardware components with a common operating environment where there is no single point of failure ■

Dr. Leon Tietz is a Professor in the Computer and Information Sciences Department at Minnesota State University, Mankato. He holds a Ph.D. in Computer Science from the University of Illinois. Prior to beginning his career in academia in 1989, he worked as a Senior Member of the Technical Staff for Texas Instruments, primarily in the area of research and development on several industrial automation projects. His interests include computer architecture and real-time embedded computing.



Look forward
to Issue 4Q 2001
on Systems Architecture